Project 1

**Brandy Bridges**

February 15, 2010

SSE 658

Dr. Paul E. MacNeil

Contents

[Part I 3](#_Toc254002044)

[Introduction 3](#_Toc254002045)

[Patterns 3](#_Toc254002046)

[Strategy Pattern 3](#_Toc254002047)

[Decorator Pattern 8](#_Toc254002048)

[Singleton Pattern 12](#_Toc254002049)

[Factory Pattern 14](#_Toc254002050)

[Part II 17](#_Toc254002051)

[Creating a Wiki 17](#_Toc254002052)

# Part I

## Introduction

For Project 1 of [SSE 658 Design Problems and Solutions](http://www.paulemacneil.com/syl658.htm) I will be picking four Design Patterns described in the Introduction through Chapter 7 of “Head First Design Patterns” and demonstrating their usefulness as well as my understanding of the patterns themselves. After I finished the reading I decided to cover the Strategy Pattern, the Decorator Pattern, the Singleton Pattern, and the Factory Pattern.

## Patterns

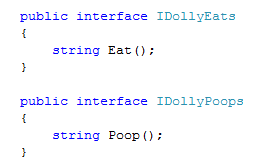
### Strategy Pattern

The text states that the Strategy Pattern “defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.” It is my understanding that using the Strategy Pattern in software engineering efforts is really doing everything you can to separate behavior in the application from the actual implementation of the application. It helps you to change an object’s behavior at runtime rather than at compile time.

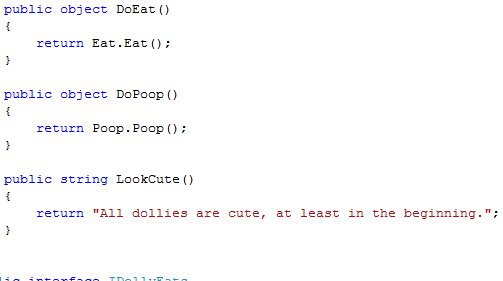
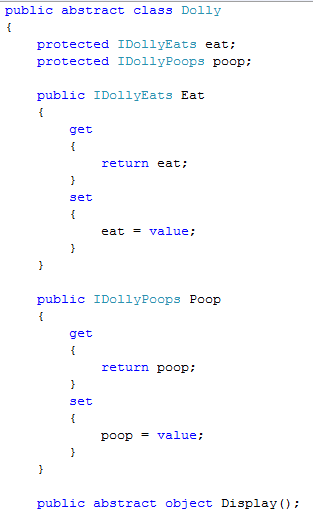
When using the Strategy Pattern, we create interfaces to handle the object behavior rather than implementing the behavior in the abstract super classes. That way we can keep adding new behaviors without affecting the behavior of existing objects.

I liked the example in the book about a duck simulator, but since I’m not interested in ducks I decided to go with a Doll simulator. I have 2 daughters and they were most helpful in helping me come up with a couple of “Dolly” behaviors that they would want to have in their favorite dolls.

First I created the initial interfaces for dolly behaviors, in this case Eat and Poop:



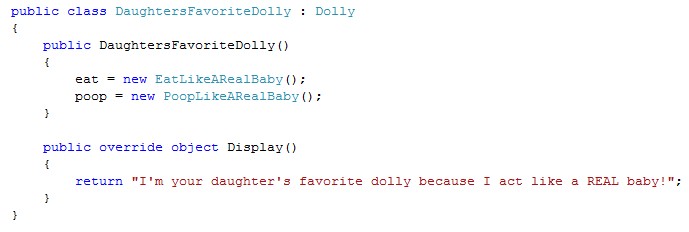
Then I created the abstract Dolly class that all classes will inherit from. I created instances of the IDollyEats and IDollyPoops interfaces that would then be accessible from all objects instantiated from this abstract class:



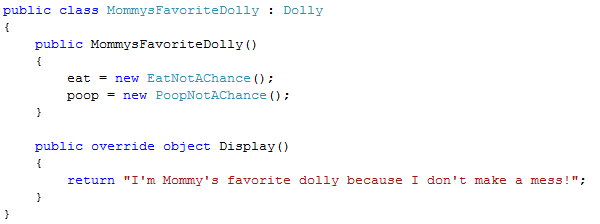
Notice that there is an abstract method defined as “Display” that must be overridden by each class that inherits from this class. I also added a method for LookCute that answers the question “Is the dolly cute?” and made sure that the returned an appropriate response. It is a concrete method that does not require override, but I could override it when I inherit from this class if I so choose.

The next step for me was to create two different classes that inherit from my abstract Dolly class and will handle all the behaviors in different ways.

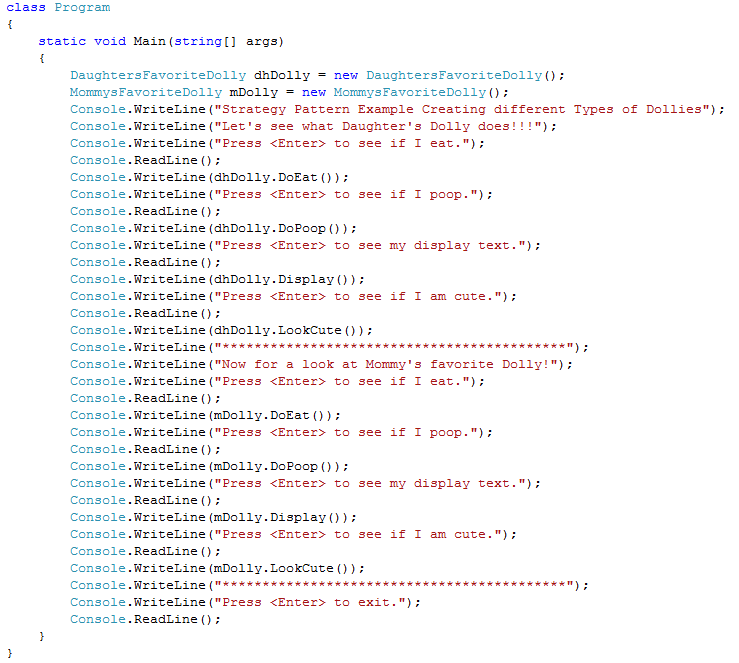
First I created DaughtersFavoriteDolly with the behaviors my daughters’ like:



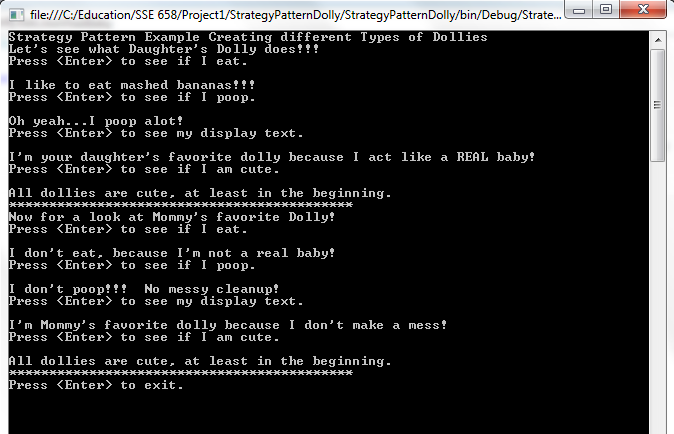
Then I created another class MommysFavoriteDolly that was more to my liking:



And my final step was actually using all the classes I’ve created. I wrote a simple .NET console application that would show all the behaviors. This is the code:



And the result is:



If I had not used the Strategy Pattern I probably still would have had an abstract class for Dolly that included an Enum to handle what kind of Dolly was being created, DaughtersFavoriteDolly or MommysFavoriteDolly then used a set of conditionals to determine behavior for each concrete class I created. I think that the ability to remove the need for these conditionals is one of my favorite benefits of the Strategy Pattern. [CodeProject](http://www.codeproject.com/KB/architecture/strategy.aspx) has a great article that outlines many Benefits and Drawbacks to using the Strategy Pattern:

#### Benefits in using Strategy Pattern

* A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
* By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
* The application can switch strategies at run-time.
* Strategy enables the clients to choose the required algorithm, without using a "switch" statement or a series of "if-else" statements.
* Data structures used for implementing the algorithm is completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.
* Strategy Pattern can be used instead of sub-classing the Context class. Inheritance hardwires the behavior with the Context and the behavior cannot be changed dynamically.
* The same Strategy object can be strategically shared between different Context objects. However, the shared Strategy object should not maintain states across invocations.

#### Drawbacks in using Strategy Pattern

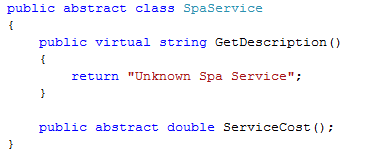
* The application must be aware of all the strategies to select the right one for the right situation.
* Strategy and Context classes may be tightly coupled. The Context must supply the relevant data to the Strategy for implementing the algorithm and sometimes, all the data passed by the Context may not be relevant to all the Concrete Strategies.
* Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviors, which some concrete Strategy classes might not implement.
* In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.
* Since, the Strategy object is created by the application in most cases; the Context has no control on lifetime of the Strategy object. However, the Context can make a local copy of the Strategy object. But, this increases the memory requirement and has a sure performance impact.

### Decorator Pattern

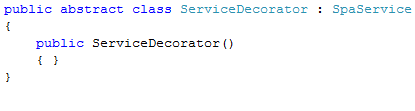
Using the Decorator Pattern can be likened to using “wrappers” for your objects. You can create the objects and then wrap them up in the same type objects to create different behaviors that are implemented at runtime. Decorators use the same interface or inherit from the same abstract class as the object they are intended to decorate. Our text describes the Decorator Pattern as an implementation that attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality of objects.

The text used an example of a coffee shop needing to calculate prices on different beverages with different condiments. It was a great example that showed how to use the Decorator pattern in a very useful way. The code I created was along the same lines, but I decided to a Spa that provides massage therapy to its clients. Different treatments are available with your massage, and these different treatments are our decorators in this case.

The first step in the code was to create an abstract class for Spa Services, with 2 methods. The first method, GetDescription() returns a default string for any SpaService requested and the second abstract method ServiceCost() insures that all subclasses must implement this method.

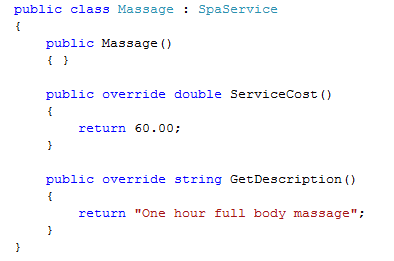


Then I created another abstract class that is a subclass of SpaService. It is basically empty except for the default constructor.

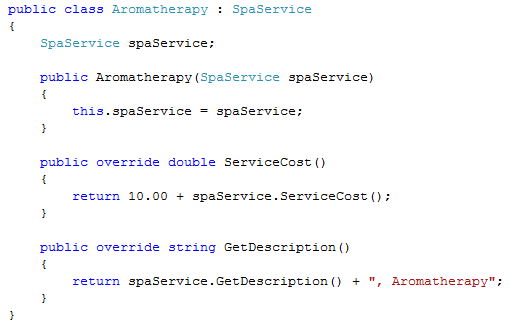


I then created four different concrete classes that are subclasses of ServiceDecorator to handle the implementations of the different Services and options.

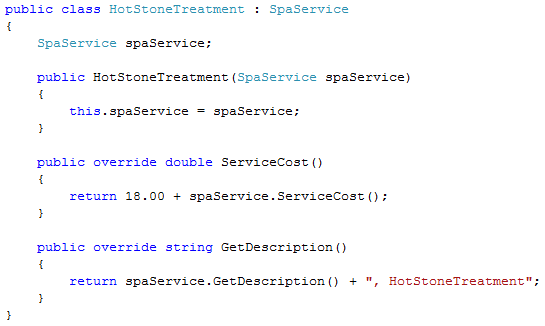
Massage:



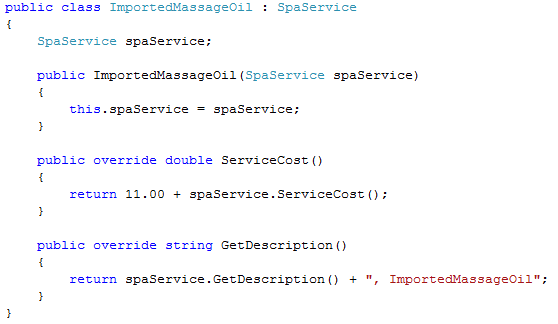
Aromatherapy:



Hot Stone Treatment:

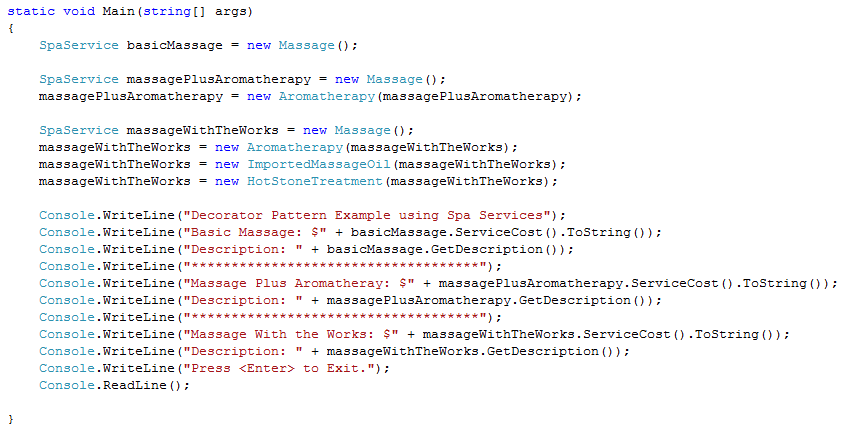


Use of Imported Massage Oil:

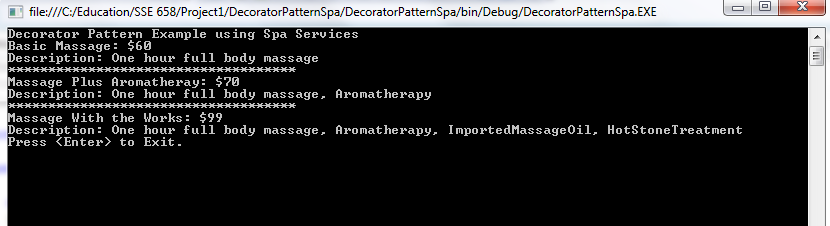


Notice that in all the decorators I created I overrode the ServiceCost() method as required by the “superclass” and also overrode the GetDescription() method to include more meaningful descriptions of the decorator being used.

The code for the actual implementation for the console application was quite simple. I just created a new spa service for each of the services that I thought would be nice if I visited a spa, then displayed the cost and descriptions associated with each service by using the methods available to the different concrete objects.



The results look like this:



Decorators provide the ability to dynamically alter the behavior of an object because a decorator can be added or removed from an object without the client knowing that a change has occurred. Using a Decorator is a good idea in situations where you may want to change the behavior of an object repeatedly (by adding and removing functionality) during runtime.

The ability to dynamically modify behavior also means that they (decorators) are useful for adapting objects to new situations without having to touch the original object's code.

### Singleton Pattern

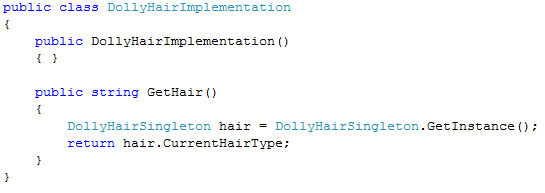
A singleton is a class which only allows a single instance of itself to be created, and usually gives simple access to that instance. Singletons are most appropriate for services that do not change based on their invocation context. A singleton can be created at startup based on context or be reset for larger scale state transitions. A lot of things fall into this category: logging, communication, database access, etc are candidates. (Most things, of course, do not fall into this category.)

Simply put, Singletons make it easy to compose a program from different components.

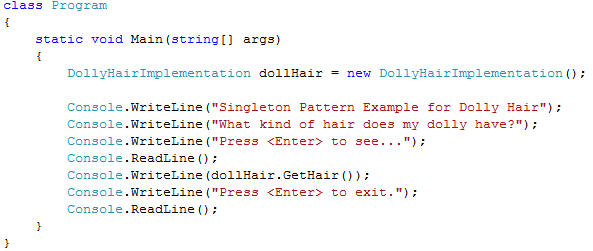
My example of a Singleton will take us back to the example of building a doll. For the Singleton example I am creating an instance of class to handle dolly hair.



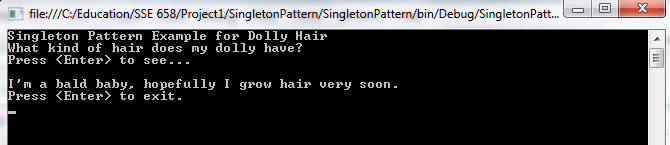
Then I created an implementation class that creates a single instance of the DollyHairSingleton class.



And finally, I created a simple console application that creates an instance of the Implementation class. I use this instance to get the information I want from the Singleton class that was created in the implementation class.



The results of my efforts are:



### Factory Pattern

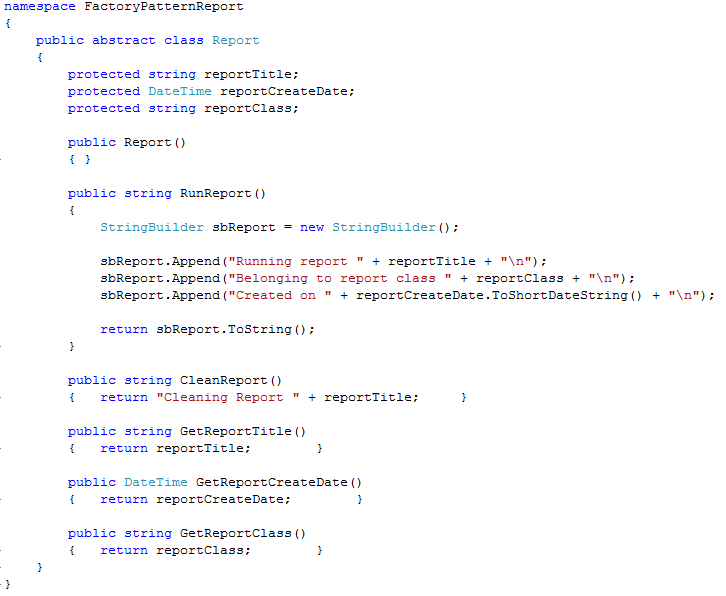
The job of the Factory design pattern is to create concrete sub classes. The essence of the Factory Pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses." Factory methods encapsulate the creation of objects. (This definition was taken from [MSDN](http://wiki.asp.net/page.aspx/310/factory/).)

Our book used the example of a Pizza Factory to demonstrate the power of the Factory pattern. We learned how to separate all the behavior that should be closed to modification from the behaviors of the different types of pizza that should be open when creating concrete classes. The common behaviors that should be closed to modification belong in the abstract super class, while the behaviors specific to each type of Pizza store belong in the concrete subclasses.

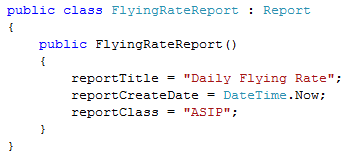
For my example code I decided to go with something that I am familiar to me. For the last year I’ve been re-creating the reports for the application that I work on in my job and we built a new architecture to handle the reports. For the purpose of this project I have decided to simplify the reports factory we created and leave out all the database results and replace it with simple strings.

We determined that all reports will have a Title, CreatedDate, and Class (not a programming class, but a class of report, like a way to logically group reports together). The behavior that all reports have are RunReport, and CleanReport (this method will clean up any staging data).

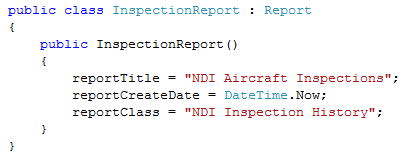
The first thing I did was create an abstract class to handle all the “same” behavior”.



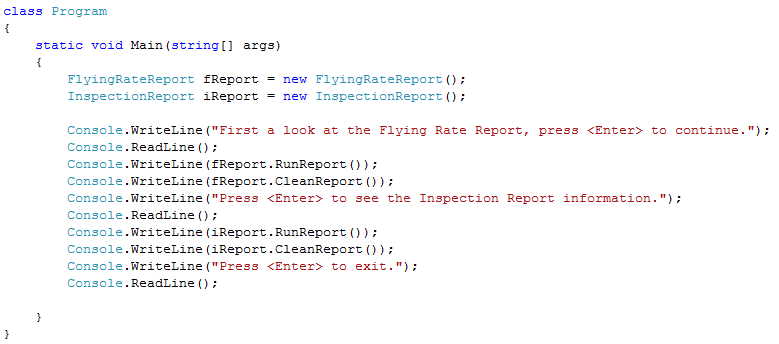
Then I created two concrete classes for the two reports I wanted to demonstrate. Each class has a different ReportClass and Title, for simplicity sake, I decided to set the CreatedDate to the current system date. The first concrete report class is for the ASIP – Daily Flying Rate Report:



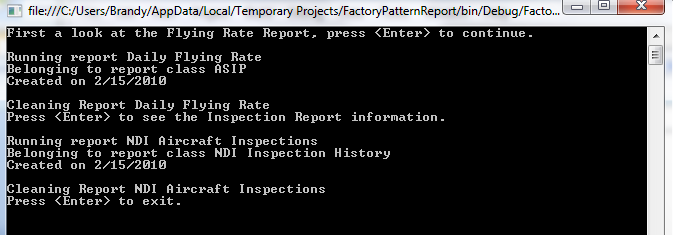
The second concrete report class is for the NDI Inspection History – NDI Aircraft Inspections Report:



I then created a simple Console application to display all the information.



Notice that I have created a new instance for each of the concrete classes that I created to handle the two different reports and I am using those instances to access the information I want to display. Also notice that each of the objects I created run the same methods (RunReport() and CleanReport()) to get the information from the objects. Both objects seem to behave in the exact same way but both give us different results. The outcomes of my efforts are:

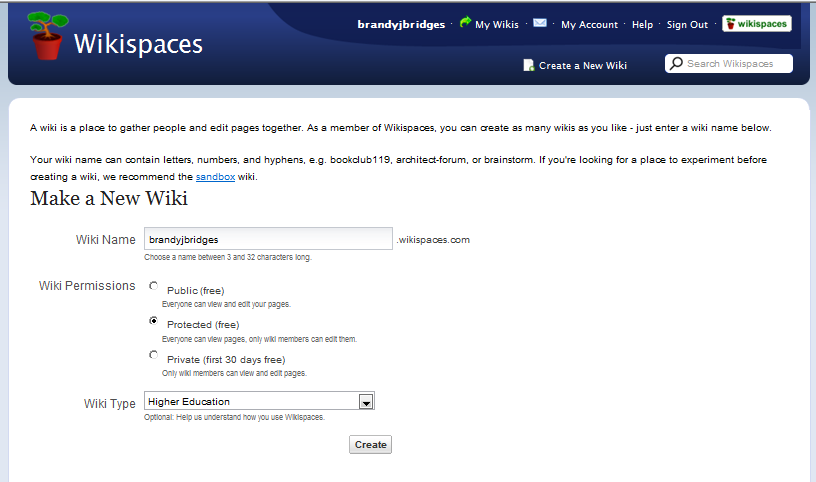


Two different reports very different results that make use of the exact same behavior defined by the same abstract class they are derived from. I built a report factory!

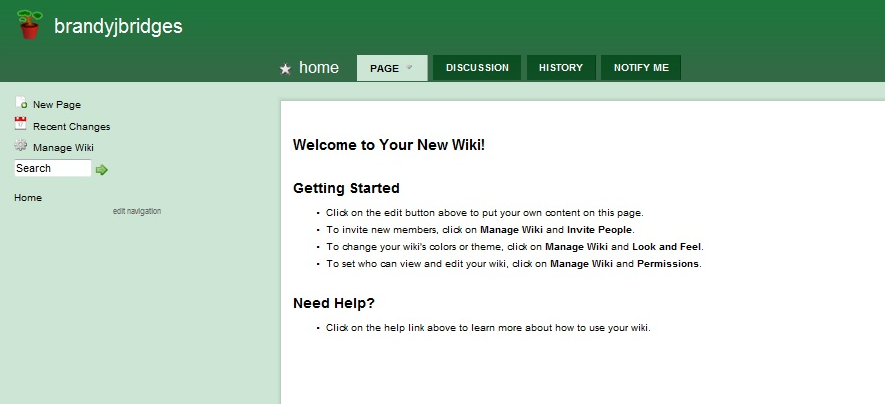
# Part II

## Creating a Wiki

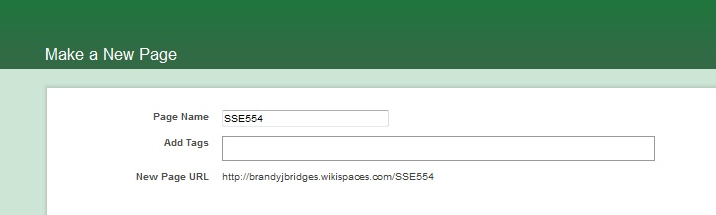
For this project I have created new wiki at <http://brandyjbridges.wikispaces.com/>, which will also be used for other classes this semester. It was very easy to create; I just registered as a new user and hit the ground running. The first thing I did was create a new wiki by clicking the “Create a New Wiki” link in completing the page that looks like this:



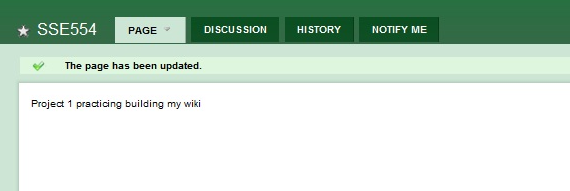
Then once I had my new wiki, I read the getting started page:



The next step was easy, I just started to add pages. I gave a little thought to the navigation and decided that I needed a page for each class I am currently taking, also a page for each project for each class. This is how I added my first page:

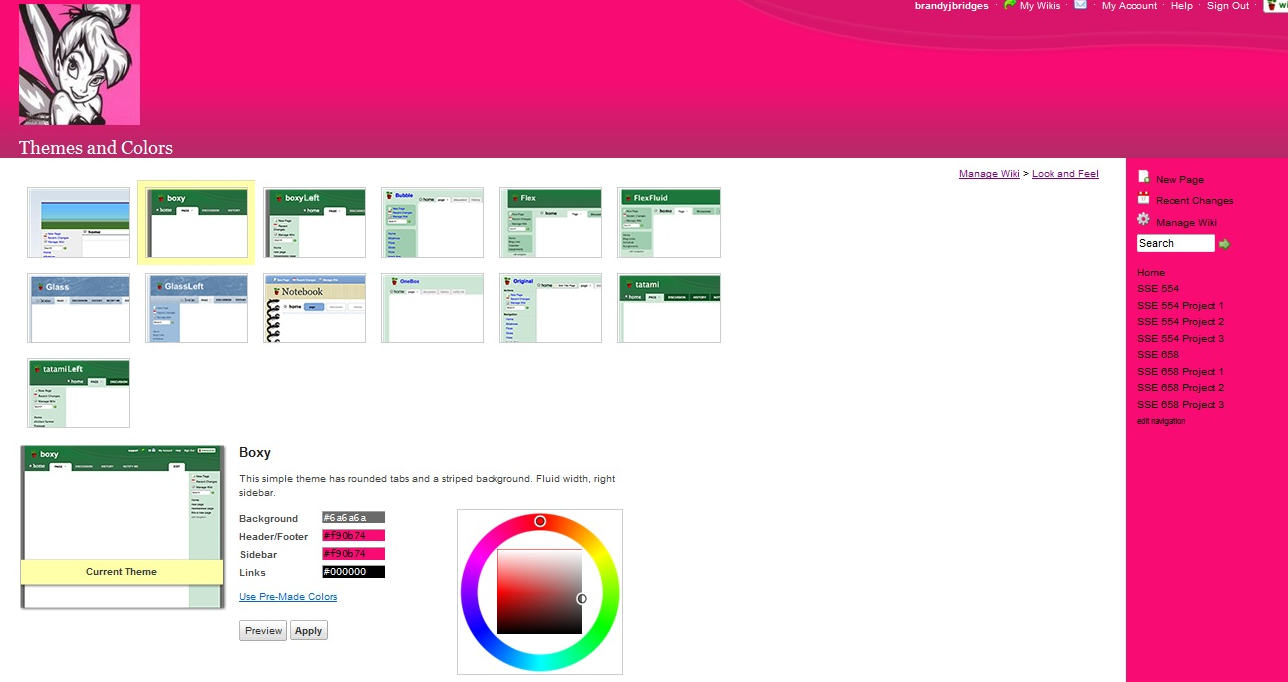


Then I added a little content as kind of a placeholder until I could come back and fill it in a little better:



Once I built the pages I wanted, I started playing with the settings to make it more my own.

* I changed the layout from the default layout.
* I changed the icon of the wiki (I like Tinkerbelle).
* I also played with the colors so that it would match my new icon a little better



I decided to upload my project to the site as well…just to practice, so this project is also available on my wiki.